commodore

comments and bulletins
concerning your
COMMODORE PET™

The Transactor

Vol. 2
BULLETIN # 5
Oct. 31, 1979

PET™ is a registered Trademark of Commodore Inc.

Case Converter

The following machine language program can be used to convert old ROM upper/lower case to the convention used by new ROM PETs.

LOAD and RUN the program below. Then LOAD the program you wish to convert for operation with new ROMs. Type SYS 826 and wella!...all upper case becomes lower case and of course vice versa.

```
100 FOR J = 826 TO 925:READ A
110 POKE J, A:NEXT
130 DATA 169,4,133,202,169,1,133,201
140 DATA 32,89,3,160,0,196,202,240,13
150 DATA 177,201,170,200,177,201,134
160 DATA 201,133,202,76,66,3,96,160,4
170 DATA 177,201,240,44,201,34,240,4
180 DATA 200,76,91,3,200,177,201,240
190 DATA 31,201,34,240,23,201,65,144
200 DATA 243,201,91,144,8,201,192,144
210 DATA 235,201,219,176,231,73,128
220 DATA 145,201,76,103,3,200,76,91,3
230 DATA 96,255,255,255,255,255,255
240 DATA 255,255,255,255,255,255,255
250 DATA 255,255,255,255,255
```

Editor's Note

Due to a delay , there will be no September issue of The Transactor. This does not mean that any less will be published or that any "double issues" will be sent to make up for lost time, but only that Transactor #5 will be dated Oct. 1979 rather than Sept. 1979. Transactor Vol. 2 subscriptions will therefore terminate the end of May rather the end of April.

The Transactor is written and produced almost entirely using the Commodore Wordprocessor. Wordpro II was written by Steve Punter of Mississauga, Ontario.

Bits and Pieces

Chuan Chee of St. Catherines, Ontario, has written the Transactor with a few items of interest:

1. When a variable is assigned the value zero with " A = 0 ", it can be substituted with " A = . ". The decimal point in this case is equivalent to zero and is 600 microseconds faster than zero. This does not mean that " 1000 " can be replaced by " 1... " since the latter is interpreted as "1" followed by a decimal point and two zeros.

2. "LIST 0" lists the whole program instead of just statement 0.

3. "(shift)RETURN" acts only as a simple CRLF instead of entering it into BASIC to be interpreted.

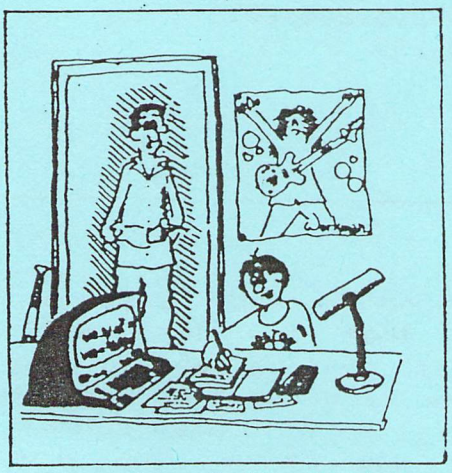
4. Statements such as " 2*-3 " and " 2/-3 " are possible on the PET whereas other computers require " 2*(-3) " and " 2/(-3) ". In fact, you can have up to 14 "-" signs and any number of "+" preceeding a numeric. Any more than 14 "-" will result in an ?OUT OF MEMORY ERROR as the stack used by BASIC is overflowed.

5. When trying ? "Y" < "YES", PET replies with -1 which is correct. Now try, A$ = "Y" : ? A$ < "YES" and PET returns a 0 which is wrong. If this is entered as a program as follows:

```
10 A$ = "Y" : ? A$ < "YES"
```

....and RUN, PET replies with -1. So why does it work in program mode but not immediate mode?

Answer anyone?



'Billy, As Soon As You Finish Your Homework Could You Help Mommy And Me Balance the Checkbook?'

'Do You Have Any "Sorry Your Program Bombed" Cards?'

Memory map: Original ROM to Upgrade ROM — Jim Butterfield

To identify a function of PET's original ROM, and/or convert it to the equivalent upgrade ROM location, use this table.

All addresses are given in hexadecimal.

OLD ADDRS	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
0000:	0000	0001	0002	000E	**	**	**	**
0008:	0011	0012	0200	0201	0202	0203	0204	0205
0010:	0206	0207	0208	0209	020A	020B	020C	020D
0018:	020E	020F	0210	0211	0212	0213	0214	0215
0020:	0216	0217	0218	0219	021A	021B	021C	021D
0028:	021E	021F	0220	0221	0222	0223	0224	0225
0030:	0226	0227	0228	0229	022A	022B	022C	022D
0038:	022E	022F	0230	0231	0232	0233	0234	0235
0040:	0236	0237	0238	0239	023A	023B	023C	023D
0048:	023E	023F	0240	0241	0242	0243	0244	0245
0050:	0246	0247	0248	0249	024A	024B	024C	024D
0058:	024E	024F	0003	0004	0005	0006	0007	0008
0060:	0009	000A	000B	000C	000D	0013	0014	0015
0068:	0016	0017	0018	0019	001A	001B	001C	001D
0070:	001E	001F	0020	0021	0022	0023	0024	0025
0078:	0026	0027	0028	0029	002A	002B	002C	002D
0080:	002E	002F	0030	0031	0032	0033	0034	0035
0088:	0036	0037	0038	0039	003A	003B	003C	003D
0090:	003E	003F	0040	0041	0042	0043	0044	0045
0098:	0046	0047	0048	0049	004A	004B	004C	004D
00A0:	004E	004F	0050	0051	0052	0053	0054	0055
00A8:	0056	0057	0058	0059	005A	005B	005C	005D
00B0:	005E	005F	0060	0061	0062	0063	0064	0065
00B8:	0066	0067	0068	0069	006A	006B	006C	006D
00C0:	006E	006F	0070	0071	0072	0073	0074	0075
00C8:	0076	0077	0078	0079	007A	007B	007C	007D
00D0:	007E	007F	0080	0081	0082	0083	0084	0085
00D8:	0086	0087	0088	0089	008A	008B	008C	**
00E0:	00C4	00C5	00C6	00C7	00C8	00C9	00CA	00CB
00E8:	00CC	00B4	00CD	00CE	00CF	00D0	00D1	00D2
00F0:	00D3	00D4	00D5	00D6	00D7	00D8	00D9	00FB
00F8:	00FC	00DA	00DB	00DC	00DD	00DE	00DF	**
0200:	008D	008E	008F	0097	0098	0099	009A	00F9
0208:	00FA	009B	009C	009D	0096	009E	009F	026F
0210:	0270	0271	0272	0273	0274	0275	0276	0277
0218:	0278	0090	0091	0092	0093	00A0	00A1	**
0220:	00A3	00A4	00A5	00A6	00A7	00A8	00A9	00AA
0228:	00AB	00E0	00E1	00E2	00E3	00E4	00E5	00E6
0230:	00E7	00E8	00E9	00EA	00EB	00EC	00ED	00EE
0238:	00EF	00F0	00F1	00F2	00F3	00F4	00F5	00F6
0240:	00F7	00F8	0251	0252	0253	.. etc.		
0260:	00AC	00AD	00AE	00AF	00B0	00B1	00B2	**
0268:	00B5	**	**	**	00B7	**	**	00B9
0270:	00BA	00BB	00BC	00BD	00BE	00BF	00C0	00C1

Memory Expansion. Cost $0.00

Ever been stuck for those few extra bytes needed to complete a program? 8K users probably know the feeling. Well now there is a consolation. If your program does not use tape file access with the second cassette, then the RAM memory devoted to the 2nd cassette buffer can be added to the memory used for BASIC.

The procedure is somewhat different for old ROMs and new but the concept is the same. Every byte of RAM in PET is physically and electronically identical. PET splits up RAM using pointers. Since these pointers are stored in RAM they can therefore be changed. Let's take a look at these pointers individually:

Old ROM:

In PETs with old ROMs, there are basically 4 pointers used to create partitions within RAM. Pointers use two bytes and are stored low order first, high order second.

1. Start of BASIC Pointer

   The start of BASIC pointer does exactly what you might think it would do; point at the start of BASIC. It is stored in locations $007A and $007B or decimal 122 and 123 and on power-up it is set to $0401 or decimal 1025. PET calls on this pointer to determine where to begin executing a RUN.

2. End of BASIC / Start of Variables Pointer

   As BASIC statements such as A=0 and XX=10 are executed, a variable table is set up immediately following the BASIC program. The variables and their corresponding values are stored in the table and and consume 7 bytes each. When called, in statements such as IF A=0 THEN... , PET jumps to the location according to the value of this pointer and begins searching. When an exact match between the variable in the current statement and one stored in the table is made, PET fetches the corresponding value and moves it to a work area and BASIC continues.

   This pointer is stored at $007C and $007D or decimal 124 and 125 and on power-up is set to $0404 or 1028 decimal. It's value, however, will constantly be changing as BASIC code is inserted or deleted. This is why the values of all variables become zero when a program change is made; if code is inserted, program text is written over the first variables in the table. If code is deleted, the bytes used by the variable table are untouched but the end of BASIC is changed and this pointer is no longer set to the start of variables.

3. End of Variables / Start of Arrays Pointer

Stored at $007E-7F or decimal 126-127, this pointer works much the same way as the previous one when array variables are called. It is also set to $0404 on power-up. As DIM statements are executed, arrays are set up starting at the location determined by this pointer. This will be the first byte following the last byte of the variable table. But what happens when a value is assigned to a new variable? If no arrays exist, the new variable and its value are simply stored in the 7 bytes following the location pointed at by the End of Variables pointer inclusive. The pointer is then updated to await the next new variable.

However, if arrays are present, a space must be created such that the new entry can be inserted as part of the variable table. This means that the arrays must first be moved up 7 bytes. Try the following:

```
        Power-up PET

    Type: TI : A=0 : TI
Note the time difference

Now type: DIM A(4,255)
     and: TI : B=0 : TI
Notice how much longer it takes
```

The extra time is spent transfering each byte of the arrays ahead by 7 bytes. Of course PET must start with the last byte of the arrays which brings us to...

4. End of Arrays / Start of Available Space Pointer

When PET must open up a space for a new variable by moving the arrays up, it calls on this pointer to determine where to start transfering bytes. PET continues this byte by byte transfer until the byte pointed at by the start of arrays pointer is also moved. The new entry is then inserted...process complete.

The End of Arrays pointer lies at $0080-81 or decimal 128-129 and also contains $0404 after power-up.

New ROM:

In new ROM PETs there are also basically 4 pointers used to section off RAM and are used the same way as old ROM PETs. However, they are stored in different places.

Pointers:	Decimal Locations:	Old ROM	New ROM
Start of BASIC		122-123	40-41
End of BASIC / Start of Variables		124-125	42-43
End of variables / Start of Arrays		126-127	44-45
End of Arrays or Start of Available Space		128-129	46-47

Moving Pointers

Now that we know where these pointers are and what they do, some experimenting can be done. Recall that on power-up the Start of BASIC Pointer is set to hex 0401 or decimal 1025. However, location 1024 is also important. It has the value zero and represents a "dummy end-of-line".

The 2nd cassette buffer starts at hex 033A or decimal 826. If this is to be included as part of BASIC memory space, the Start of BASIC Pointer must be moved DOWN. Since location 826 will have to serve as the dummy end-of-line character, the new start of BASIC will be 827 or $033B. The procedure is as follows:

```
POKE 826 , 0     :Dummy end-of-line
POKE 122 , 59    :low order byte of pointer = $3B (3*16+11)
POKE 123 , 3     :high order byte = $03
(New ROM users will substitute the other POKE locations.)
```

That takes care of the Start of BASIC Pointer but all those other pointers are still up where they used to be when BASIC started at $0401. They must also be moved down. We could use POKE to accomplish this however a NEW command will do them all at once. Therefore execute a NEW and then print FRE(0). You should be returned 7362 bytes free, an increase of 195 bytes! This may not seem like much but when those few extra bytes are needed to add those finishing touches it could come in very handy.

Now that the BASIC memory space has been increased does not mean that your program will automatically fill up this space. Besides, the NEW command removes your program anyways. One way to effectively use this modification is the following:

1. Power-up and LOAD your program.
2. Using UNLIST (described in Transactor #2, Vol. 2), record the program.
3. Increase memory using the steps outlined above, and...
4. Using the Merge procedure, also described in Transactor #2, bring the program back in by essentially merging it with empty space.

Now the first 195 bytes of your program will be resident in what used to be the second cassette buffer. Remember, you no longer have a second cassette buffer until you either reset the machine or re-adjust the pointers so don't try to use it or your program will be clobbered!

Sooner or later you will need to SAVE the program. However, this can no longer be done in the conventional manner. Take a look at the method used by Bill Seiler on the second page of Transactor #3. Execute lines 100 through 220 directly on the screen exactly as shown. This is a modified SAVE. The SYS63153 accesses the tape write routines in ROM.

Now that a recording has been made there is one last problem. When the program is LOADed back into the PET, the Start of BASIC Pointer is not automatically set. It stays at 0400 but our program starts at 033A. POKE 122,59 and POKE 123,3 will fix this up.

A Short Note on Tapes

When a program is recorded on tape, the start and end addresses of that program are also recorded as part of the tape header. Therefore, when the program is LOADed, PET first looks at the start address and begins transfering bytes from tape into RAM. The first byte is transfered to the location specified by the start address. Increasing your memory using this method does NOT mean that your programs will LOAD to this extra space. However, they can be modified to do so. The information needed is in the article by Jim Butterfield on the first page of the first Transactor in Vol. 2.

Cassette file END markers

Jim Butterfield, Toronto

End-of-tape blocks may be written on cassette tape.
If the computer is searching for a program or file and finds this block, it will stop and report ?FILE NOT FOUND.

This is useful since it saves having to search through yards of empty tape.

Here's how to write such an END marker. When you're saving the last program on a tape, use SAVE "PROGRAM-NAME",1,2. The END block will be written behind the program. If you are writing files, open the last file with OPEN 1,1,2,"FILE-NAME". The END block will be written after the file is closed. In either case, it's the value 2 which triggers writing of the END marker.

If you need to write an END marker on tape without having a program or file to write, you can do it with one of the following statements:

Original ROM: POKE 241,1 : POKE 634,5 : SYS 63673
Upgrade ROM: POKE 212,1 : POKE 634,5 : SYS 63622

The first POKE statement specifies cassette drive #1. You may change it for cassette #2 if you wish.

Attention Multi-Peripheral Users-

It has been found that when more than one peripheral is connected to the IEEE-488 buss, a slight problem may occur should one device be ON and the other OFF. Take for example the following sequence of events.

PET ON
Printer OFF
Disk OFF

Type: OPEN 1 , 8 , 4 , " 0:DISKFILE , S , W "

PET responds: ?DEVICE NOT PRESENT ERROR

This is of course what you would expect. Now power up the Printer, leaving the disk unit OFF.

Type: OPEN 1 , 8 , 4 , " 0:DISKFILE , S , W "

PET responds: READY.

But the disk is OFF or essentially "NOT PRESENT". Therefore:

PRINT#1,"FILE DATA"

...will result in lost data.

There is, however, a test that can be made to protect against lost info. The status word, ST, is set to -128 whenever the above situation occurs. Therefore the following test could be included immediately after the OPEN statement:

IF ST < 0 THEN PRINT "DEVICE NOT PRESENT"

Don't be alarmed since any programs using disk file access are usually loaded from the disk, the disk will be turned ON anyways and the above situation will probably never be encountered.

Frans VanBurmen
Toronto
30Sep79

?LOAD ERROR

This note deals with program load errors on the 8K PET (Release 1), and how to recover from them.

Within two days after setting my PET (Nov78), I discovered the merits of back-up copies of programs and data files.
All I did was press PLAY and RECORD when the message said to press PLAY! It was only some twenty seconds, but it was sufficient to wipe out the file header and make the file inaccessible.

Ever since I've made sure to keep multiple copies, on the same tape for programs under development, on a dedicated back-up tape for programs that are more or less static.
So also the Journal program that I was developing back in July. The only thing was, I was also working on another program, which that I accidently saved on the wrong tape. Scratch Journal version 0.6.

No real harm done, since I still had version 0.5, right? Wrong! It just so happened that good old 0.5 had a load error.
I tried just about every thing, demagnetize & clean heads, both tape drives on my PET, LOAD vs SHIFT/shift, freeze cassette, rewind tape evenly, loosen screws in cassette housing and play on several other PETs. About the only thing I did not play with was head allignment (since the tape had been written with this allignment, it ought to be optimal for reading).

All to no avail. A load error I got, and load errors I kept on getting.
Yet I knew the data was there! There were some 3500 characters on that tape, most of which loaded correctly, but could not LIST, RUN or SAVE.

Since I still needed the Journal program, my choice was simple: salvage or re-develop and re-enter from memory.
So, with an ingenuity born of laziness (that being one of the prime qualifications for all programmers), I salvaged!

From Jim Butterfield's memory map (see The Transactor 9 vol 1 -or The Best of Transactor vol 1, pp 149-155 - and vol 2 #3) and my own disassembled listing of ROM, I had since acquired essential information on pointer fields and routines.

First let me introduce the cast of characters:

- the program, it starts at loc 1024
- the file header, at loc 634 for tape 1, loc 826 for tape 2
- the load start point in the file header at offset 44
- the load end point in the header at offset 46
- the start of BASIC pointer at loc 122
- the end of BASIC/start of variables pointer at loc 124
- the end of variables pointer at 126
- the start of available space pointer at loc 128
- the Next Instruction Pointer (NIP) that precedes ever BASIC program instruction

.the BASIC Line Number (BLN) that is part of every statement
.the zero byte that identifies the end of each BASIC statement
.the End Of Program (EOP) marker, which is a dummy NIP of which at least the second byte contains zero.

After a normal load PET updates the end of BASIC pointer, the end of variables and the start of available space pointers based on the end of load address from the file header.
Not so on a load error, the end of BASIC/start of variables pointer remains at 1024 (the start of BASIC pointer to be exact).
However, if variables are used they will be stored starting location 1024, i.e. smack on top of the program.
The following code will fix that (assuming LOAD from tape #1):

```
?Pe(637);Pe(638)  - which results in the values being printed
                    (remember, no variables may be used yet
 237 17             example (237+256*17=4589)

Po124,237:Po125,17- set end of BASIC/start of variables
Po126,237:Po127,17- end of variables
Po128,237:Po129,17- start of available space.
```

Whew! Now we can use variables, since they will now be stored starting at 4589.

Next step is to rebuild the NIP pointer chain, where the NIP preceding every BASIC statement points to the NIP before the next statement, until we get to the dummy NIP that marks end of program.
SYS 50224 is an operating system routine that does just that. However, it does that based on zero bytes. It assumes that every zero byte it encounters represents either the end of a statement or the end of the entire program. Thus if the load error introduces spurious zeroes, they may throw SYS 50224 for a loop, and the routine would store NIPs on top of valid data. If it does work, however, it's the by far easier method. If it does not work just reset the system and try the other possible approach.

The alternative is to write a one-line immediate routine that will follow the existing chain as far as possible, fix and continue.
The following routine will print a list of NIPs in ascending order, with line numbers (BLN), also in ascending order.
Any irregularity in either list indicates a load error.

```
I=1025    initialize pointer to first NIP

FoK=1TO900:J=Pe(I)+256*Pe(I+1):B=Pe(I+2)+256*Pe(I+3):
  ?I,J,B:I=J:Ne
```

This results in a list such as:

```
1025    1052    10
1052    1066    20
1066    1099    21
1099    1120    50
1120    1156    60
```

Screen Print Routine

The following is a machine language subroutine that will copy the contents of the screen onto 2022/23 printers. It resides in the second cassette buffer and could be incorporated very neatly into any BASIC program where a hard copy of the screen might be required.

```
                    ; SCREEN PRINT ROUTINE
                    ; CALL WITH SYS 826
033A                        *=   $033A
033A                POINT   =    $1F
033A                RFLAG   =    $21
033A                COUNT   =    $22
033A                CR      =    $0D
033A                DEVICE  =    $D4
033A                CMD     =    $B0
033A                PRINT   =    $FFD2
033A                SLISTN  =    $F0BA              ;LISTEN TO IEEE
033A                ATNOFF  =    $F12D
033A                BUSOFF  =    $FFCC
033A                SCREEN  =    $8000
033A                CASE    =    $E84C,             ;GRAPHICS OR LC
033A A9 80          SCPRT   LDA  #>SCREEN
033C 85 20                  STA  POINT+1
033E A9 00                  LDA  #<SCREEN           ;SET POINTER TO
0340 85 1F                  STA  POINT              ;START OF SCREEN
0342 A9 04                  LDA  #4
0344 85 B0                  STA  CMD
0346 85 D4                  STA  DEVICE
0348 20 BA F0               JSR  SLISTN
034B 20 2D F1               JSR  ATNOFF             ;OPEN PRINTER
034E A9 19                  LDA  #25                ;25 LINES
0350 85 22                  STA  COUNT
0352 A9 0D          LINE    LDA  #CR                ;START NEW LINE
0354 85 21                  STA  RFLAG              ;RVS-OFF
0356 20 D2 FF               JSR  PRINT
0359 A9 11                  LDA  ##$11              ;SHIFT FOR L/C
035B AE 4C E8               LDX  CASE
035E E0 0C                  CPX  #12
0360 D0 02                  BNE  LOWER
0362 A9 91                  LDA  ##$91              ;SHIFT FOR GRAPHICS
0364 20 D2 FF LOWER         JSR  PRINT
0367 A0 00                  LDY  #0
0369 B1 1F          MORE    LDA  (POINT),Y          ;SCREEN CHAR
036B 29 7F                  AND  #$7F               ;STRIP RVS
036D AA                     TAX                     ;STORE
036E B1 1F                  LDA  (POINT),Y          ;CHECK RVS
0370 45 21                  EOR  RFLAG              ;SAME AS LAST CHRPRIN
0372 10 0B                  BPL  SAME
0374 B1 1F                  LDA  (POINT),Y
0376 85 21                  STA  RFLAG              ;LOG NEW RVS STATUS
0378 29 80                  AND  #$80
037A 49 92                  EOR  #$92               ;BUILD RVS ON/OFF
037C 20 D2 FF               JSR  PRINT
037F 8A             SAME    TXA                     ;RECALL PRINT CHAR
0380 C9 20                  CMP  #$20
0382 B0 04                  BCS  NOTALF
0384 09 40                  ORA  #$40               ;CHANGE ALPHA ZONE
0386 D0 0E                  BNE  SEND               ;BRANCH ALWAYS
```

```
0388 C9 40     NOTALF   CMP  #$40
038A 90 0A              BCC  SEND
038C C9 60              CMP  #$60
038E B0 04              BCS  GRAPH
0390 09 80              ORA  #$80
0392 D0 02              BNE  SEND                ;EXPAND- ALWAYS
0394 49 C0     GRAPH    EOR  #$C0
0396 20 D2 FF  SEND     JSR  PRINT               ;PRINT CHAR
0399 C8                 INY
039A C0 28              CPY  #40                 ;LINE FINISHED?PRINT
039C 90 CB              BCC  MORE                ;NO, DO IT AGAIN
039E A5 1F              LDA  POINT
03A0 69 27              ADC  #39                 ;YES, MOVE SCREEN POINTER
03A2 85 1F              STA  POINT               ;TO NEXT LINE
03A4 90 02              BCC  *+4
03A6 E6 20              INC  POINT+1
03A8 C6 22              DEC  COUNT               ;ONE LESS LINE
03AA D0 A6              BNE  LINE                ;GO FOR ANOTHER
03AC A9 0D              LDA  #CR
03AE 20 D2 FF           JSR  PRINT
03B1 4C CC FF           JMP  $FFCC               ;CLEAR BUS ; QUIT
```

```
90 REM BASIC LOADER FOR SCREEN PRINT ROUTINE
100 FOR J = 826 TO 947
110 READ A : POKE J , A
120 NEXT
200 DATA 169,128,133,32,169,0,133,31
210 DATA 169,4,133,176,133,212,32,186
220 DATA 240,32,45,241,169,25,133,34
230 DATA 169,13,133 33,32 210,255,169
240 DATA 17,174,76,232,224 12,208,2
250 DATA 169,145,32,210,255,160,0,177
260 DATA 31,41,127,170 177,31,69,33 16
270 DATA 11,177,31,133,33,41,123,73
280 DATA 146,32,210,255,136,201,32
290 DATA 176,4,9,64,208,14 201,64,144
300 DATA 10,201,96,176,4,9,126,208,2
310 DATA 73,132,32,210,255,200 192,40
320 DATA 144,203,165,31,105,39 133,31
330 DATA 144,2,230 32,198,34 208 166
340 DATA 169,13,32,210,255 76,204,255
```

```
 1156   585    70
 585    126652 12445
BRK
```

Clearly 1156,1157 do not contain a valid NIP.
In this specific instance it appears that 1156,1157 are indeed the NIP (since the BLN looks to be correct), but the NIP has been clobbered due to the load error.
Frequently load errors are a result of timing errors.
This is where the read routine cannot handle the variations in tape speed that it perceives.
The result is commonly that the read routine reads more bits than were actually written to the tape. Conversely the routine may actually read fewer.

In my case the errors occasionally were wrong characters, or in some instances one or more characters missing or extra. Yet subsequent characters would still by and large be correct. In other words, it would appear that the read routine can recognize and synchronize with byte boundaries as recorded on tape.

The important thing here is that frequently a NIP address would be out by plus or minus one or two bytes, but so would the next one and the next.

To view what the internal representation of the program looks like, an immediate routine such as the following may be used:

```
I=1155    -loc of last valid(?) NIP, minus 1 to check
           for presence of preceding zero

FOR=ITOI+60:?PEEK(I);:NEXT - would result in

 0  132  4  70  0  145  137  32  49  48  48  44  50 48 ..
      NIP    BLN    ON  GOTO      1   0   0   ,   2  0 ..
```

(sorry, not the interpretation shown on the second line)

An other approach is to print the location number as well as its content. That makes it much easier to see what is going on:

```
FOR=ITOI+60:?'R'K'r'PEEK(K);:NEXT
```

```
    'R' - Reverse video on
    'r' - Reverse video off
```

This would show alternately a location address (in reverse video), followed by it content:

```
1055  0  1056  132  1057  4  1058  4 ..... 1072  0
    1073  156  1074  4 ...
```

This facilitates checking the NIP actual location against the expected one (as contained in the preceding NIP).

A further variation on this to include two cursor-left characters:

```
FoK=ITOI+60:?'R'K'rcl'Pe(K)'cl';:Ne
```

cl - a single cursor-left character

This gets rid of the cursor-right the PET inserts after all numbers. Not only does it compress the listing, it also allows reuse of the statement (such as after a POKE, or for a different area) without occasional digits from the previous data showing through.

If an individual NIP is wrong, the most expedient solution is to POKE in a new value.
If, however, several subsequent NIPs are all out by the same amount, moving over the rest of the program may be indicated.
Visual inspection will have to indicate which bytes to surpress, or where to open it up.
Remember the main concern right now is to get the program in such shape that it can be LISTed and updated normally.

On compression, as in the following routine, bytes are copied into lower numbered locations. Thus if location 1112 is stored in 1111, 1113 in 1112, 1114 in 1113, etc., location 1112 has already been used by the time 1113 is stored into it, and thus may be safely clobbered
For example:

```
FoI=1111TO4589:J=Pe(I+2):PoI,J:Ne
```

The +2 in the PEEK command causes everything to be moved over ('to the left') by two bytes.

Note that merely changing the +2 to -2 will not move everything two positions to the right.
Instead the leftmost two characters will be propagated through the entire section being moved. In the above example (with the +2 changed to -2) byte 1111 would be picked up first, and stored in 1113. Then 1112 would be stored in 1114. Next 1113 would be picked up to be stored in 1115. But 1113 contains the value from 1111 by now, and that is what would be deposited in 1115. Thus 1111 ends up in 1113, 1115, 1117, etc., with 1112 ending up in all the inbetween locations.

To handle such a shift right properly, the move has to start from the right, e.g.:

```
FoI=4589TO1111STEP-1:J=Pe(I-2):PoI,J:Ne
```

That essentially sums up the totality of this technique for salvaging programs from load errors.
I do, however, sincerely hope that you'll never have to use it.

The IEEE-488 Bus

A parallel interface designed to exchange data with selected devices connected to the bus.

Many devices may be connected at the same time, but only the one that has been selected will send or receive data. For example, two printers and a disk unit could be connected to a bus; the Basic program would arrange to send to or receive from the various devices as desired.

Selection works by means of a "calling" system. Before sending data, the computer first sends a selection character, which commands the appropriate device to "listen". If the device is connected, it will acknowledge the command. Now the data is sent; each byte is acknowledged by the receiving device. Finally, the device is disconnected by an "unlisten" command. To receive data, the computer instructs the appropriate device to "talk". It then accepts data until the device signals "end of data", at which time the computer sends an "untalk" command.

Commands are distinguished from data by using a special line called ATN (attention). If the ATN signal is low (meaning true), the information being sent is a command: talk, untalk, listen, or unlisten. If the ATN signal is high (meaning false), the information being sent or received is data. In this system, only one direction is used: the computer sends ATN and the devices receive it. When ATN is low, all devices receive the commands, to see if they are being selected. When ATN is high, only the selected device will accept data.

Another line, called EOI (end or identify) is used to signal the last byte of data. It works in both directions: if the computer is sending, it signals EOI low (meaning true) with its last character; if the device is sending, it signals EOI low if it has no more data after the character it is sending.

When a device sends to the computer, it delivers each character only when invited by the computer. Similarly, the sending computer delivers characters only as fast as the device is ready for them. This flow is controlled by a "handshake" procedure.

An example of selection: When Basic executes OPEN 3,4, the IEEE-488 bus sets the ATN signal low and transmits hexadecimal 24 to the data lines, instructing device #4 to listen. If the device does not answer, Basic will return either DEVICE NOT PRESENT (ST=128 decimal) or WRITE TIMEOUT (ST=1). Subsequently, when the command PRINT#3,"HELLO" is given, the ATN signal is again set low and hex 24 transmitted to instruct #4 to listen; then ATN is set high, and the characters H, E, L, L and O are sent, with EOI set low during the transmission of the O character; finally, the ATN is set low and hex 3F is sent to cause the device to unlisten. Note that we haven't closed the file yet; but we have (temporarily) disconnected the device.

Using CMD on the IEEE-488 Bus

CMD does two things:

- --it opens the appropriate device to "listen";
- --it will divert output, normally directed to the screen, to the IEEE-488 bus.

Both CMD activities are cancelled in any of three ways:

- --preferred: when the bus is addressed with a normal PRINT# command;
- --when any INPUT or GET is performed;
- --when a Basic error is encountered.

It is best to avoid CMD within Basic programs, since any use of INPUT or GET will cancel it, and the programmer will have to arrange to repeat the CMD as necessary. Use PRINT# wherever possible. CMD is most useful in obtaining program listings. The preferred method:

```
OPEN 4,4      (identify the printer as device # 4)
CMD 4         (open the printer to listen & redirect output)
LIST          (do the listing)
PRINT#4       (cancel the CMD functions)
CLOSE 4       (close the file)
```

Never close a file until you have first cancelled the CMD command.

IEEE-488 Handshake: a brief technical description

The same handshake procedure is used for both command and data transmission.

The talker uses the DAV (Data available) line to indicate that valid data has now been placed on the bus. The listener uses two lines: NRFD (Not ready for data) to indicate that it is not yet willing to receive data; and NDAC (Data not accepted) to indicate that it has not yet taken data from the bus.

Transfer of data takes place in the following manner:

1. The talker initially places DAV high (meaning false) to indicate that data is not being sent yet. The listener will have NDAC low (meaning true) to indicate that no data is being received. If the listener is still working on something (say, printing the previous character) and can't accept data yet, it will set NRFD to low (true), meaning it's not ready.

2. The talker checks the NRFD and NDAC lines for both high (meaning false). If they are both high, something is wrong. If the computer is the talker, it will send DEVICE NOT PRESENT.

3. The talker places its data on the bus, but doesn't signal DAV low for data available until it sees the listener's NRFD is high, which signals that the listener is ready to receive data. The talker will wait forever - there is no timeout.

4. The data is ready, so the listener accepts and stores it. Then the listener sets NRFD low (true) and NDAC high (false) to acknowledge its receipt. The listener has a time limit on this activity: if it doesn't complete in 64 milliseconds, the talker will flag TIMEOUT ON WRITE.

5. The talker responds to the acknowledgement by setting DAV high, meaning that the data is no longer offered, and then clearing the data bus.

6. The listener detects the change in DAV, and realizes that its acknowledgement has been seen. It returns NDAC to low, completing the character exchange cycle. There is a time limit here: if the listener doesn't see DAV go high within 64 milliseconds, it will flag TIMEOUT ON READ.

Frans VanDuinen
Toronto
27Sep79

Delete Rest of Instructions in Program

One of the more exciting, albeit undocumented, instructions on the PET is the 'Delete Rest of Instructionsin Program' or DRIP instruction.
If you haven't yet had occasion to use it, consider yourself lucky.

Under certain conditions the updating and replacing of a BASIC program instruction results in the disappearance of that and all subsequent instructions in the program. As this seems to happen only after extensive (and not as yet saved) program changes have been made, the result is a lot of excitement.

This note describes what happens, when, how to recover from it, and covers a technique that seems to prevent it, but since I'm not sure how or why I can't be certain that the preventative measure always works.
The content of the note applies to Release 1 of the PET 8K system, the 'old ROM'.

The only cause that I am certain about is an interupt of a program occurs that is using the PRINT# to write to the IEEE bus. (Where my printer sits as device no 4.)
Any subsequent attempt to change the program frequently results in a 'DRIP'.
However, if I enter a 'CLR' command in between or cause an error, such as a RUN command with an invalid operand, a DRIP does not arize.

The symptoms are as follows. BASIC does somehow not recognize that the newly entered (updated) statement matches an existing number. BASIC therefore treats the updated instruction as a new one, and moves over the rest of the program to make room to insert this 'new' instruction.

However, BASIC makes other errors, that are even more severe. It inserts a zero in the high-order (second) position of the Next Instruction Pointer (NIP) of the first occurrence of the updated instruction, thus signalling the end of Program. The part of the program that has been moved to allow for the insert of the 'new' instruction, has not had its pointers updated.

Fortunately, BASIC leaves the 'end of BASIC/start of variable' pointer intact, so variables can be used.

The solution of this problem is actually quite simple:

- remove the spurious zero
- rebuild the pointerchain.

I had visions of sophisticated program logic to reconstruct pointers based on the minimum and maximum number of bytes per instruction, zero bytes, relationships between statement numbers and visual inspection.
But once more, Jim Butterfield to the rescue! His list of routines identifies one called 'Corrects the chaining

between BASIC lines after insert/delete !!!

As it turns out, it is very simple: if the address pointed by the current NIP, which itself is a NIP, contains a zero in the second byte, it is considered to be the end of program All other zeros starting at NIP+4 (to make allowance for the BASIC line number)are considered to represent the end of an instruction.

Thus by removing the zero that erroneously flags End Of Program, the pointer chain can be rebuilt by invoking this routine (SYS 50224).

Theoretically SYS 50224 could also be used to find the location on the End OF Program zero byte, as it leaves the address of the last NIP in locations 113,114.
Unfortunately, however, this is not a closed subroutine. It terminates by branching (JMP) into the PET's main command processing logic, rather than returning to the caller.
Locations 113,114 have been clobbered by the time control is returned to the Keyboard

What can be used is an immediate command,such as:

```
I=1025:FOR=1TO1000:J=PE(I)+256*PE(I+1):?I,J:I=J:NE
```

which will print a list of NIPs, that is in ascending order, upto and including the address of the faulty NIP, e.g.:

```
  3255      3272
  3272      3301
  3301      3356
  3356      55
  55        12356
BRK (stop as soon as dir occurs)
```

In this example locations 3356,3357 contain the faulty NIP. (These bytes contain 55 and 0 respectively.)
Now all that is required is the following:

```
   POKE 3357,1
   SYS  50224
```

The POKE instruction eradicates the value zero, and the SYS rebuilds the pointer chain.

In above example byte 3356 would originally have contained 13 (13*256+55=3383), however that is immaterial as the instruction that was there has been moved, while the SYS 50224 only makes the distinction zero or non-zero.

I hope this will allow others to deal with the DRIP instruction, however, the approach of frequent saving of program updates is still preferable!

Cross Reference

I've recently completed a cross reference of all instructions in the 8K PET system (Release 1) that reference individual RAM and PIA/VIA locations.

The process was actually quite simple. I used a disassembler that scans for selected opcodes and specific operand addresses. It only took some 280 hours of processing to scan the system the required few hundred times.

Because of its size I've only included the first page of the approximately 25 page listing.
The tape file for this cross-reference contains some 77000 characters and 3000 records.

If anyone is interested, I'll gladly send them a copy of either the listing or the tape.
I must, however, insist on a copying, postage and handling charge. Photocopies cost me a dime (actually 9 cents) like anybody else, and it takes over an hour to copy the tape file.

If you are interested, send me a note at below address, and include $4.50 for photo copies, $ 6.00 for the tape (including a print/display program), or $10.00 for both copies and tape. For the photocopies specify decimal or hexadecimal addresses.

Frans VanDuinen
175 Westminster ave.
Toronto, Ont. M6R 1N9

```
CROSS REFERENCE LISTING     PAGE 3

     LEGEND
     notation preceding address
      A  - Accumulator
      M  - Memory
      X  - X-register
      Y  - Y-register
      F  - status flags
      S  - stack register

     notation following address
      I  - Indirect addressing
      XI - Indexed indirect (on X reg)
      IY - Indirect indexed (on Y reg)
      X  - Indexed on X reg
      Y  - Indexed on Y reg
      D  - Dynamically modified instruction addr

     0  L=3  Jump instr to USR routine
       byte 0=$4C - JMP
          A C493 X    A C4C8 X    A C4F8 X    A C516 X    M D3B7 X    M D891 XI
          M E0B0 XI   M E0DB      A E199 IY   M FD48      M FD54 IY   M FD5F
          F FD6B IY   M FE05      A FE0D IY   M FF11 IY   F FE13 IY   X FF43
          A FFB2 XI

     1  Jump instr to USR routine
          M D3BB X    M D773 X    A D7A0 Y    A D7A3 X    Y D866 X    M D86C X
          M D87B X    M D87F X    M D881 X    M D883 X    M E0E1      M FD4A
          A FD50      M FD59      F FD5B      M FD61      A FD67      M FD72
          F FD74      M FE09      M FE18      F FE1A

     2  Jump instr to USR routine
          M D3BF X    A D799 Y    A D79C X    Y D862 X    M D868 X    M D885 X
          M E0E3      A FD52      A FD69      M FE02      A FE0F      M FF2D
          F FE2F

     3  L=1  Active I/O device no (for prompt suppress)
          A C35B      M C364      Y C47C      M C993      A C9CE      A C9D2
          A C9E2      A CA5E      A CA6A      A CA84      M CAB4      X CAC1
          M CAD1      A CAD6      M CADD      A CAFB      A CB10      A CB17
          A CB5D      A CC06      A D792 Y    A D795 X    Y D85E X    M D864 X
          M D887 X    M E0F2

     4  L=1  Nulls for carriage return & linefeed (not used)
          M C76A      X C9E8      A D78B Y    A D78F X    Y D85A X    M D860 X
          M D889 X    M E0F6

     5  L=1  Cursor pos'n for INPUT & PRINT (column where next ch
     ar will go)
          M C4DE Y    A C4E1 Y    M C501 Y    M C9D6      M C9F4      A C9F9
          A CA14      A CA5A      M CA62      Y D285      M F0F8      M F1CF
          M E1D4

     6  L=1  Terminal width (not used)
```

Technical comment on FOR/NEXT loop structures. Jim Butterfield, Toronto.

Recent remarks on popular Basic implementaxtions indicate that difficulties may be encountered if the programmer jumps out of a FOR/NEXT loop.

This would be very serious if true. The programmer doing a table search would be required to continue scanning the table even after finding the item he wants; or to use questionable practices such as meddling with the loop variable while still within the loop.

Fortunately, it's true only for a few complex situations - and these are easy to fix if you understand how the dynamic FOR/NEXT loop works. (Dynamic loops are those set up during an actual program run, as contrasted to pre-compiled loops which are checked out before the actual run starts).

When a dynamic interpreter, such as Microsoft Basic, encounters a statement such as FOR J= ... it sets up internal tables to manage the loop. These internal tables contain such things as: where to return if a NEXT J is encountered; the identity of the loop variable (in this case, J); whether the loop is counting up or down, etc.

These tables will remain until one of three things happens. If the loop goes through its complete range (by encountering a suitable number of NEXT J statements); if a new FOR J statement is found; or if a higher priority loop is terminated for either of the previous reasons.

The last rule is very sensible, and it's worth a closer look. Suppose we have set up a sequence of commands such as: FOR I= ... : FOR J= ... : FOR K= ..., and suppose the computer, while dealing with these three loops, finds a new FOR I=... statement. It very wisely says, in its own computerese, "OK - looks like the big loop is being restarted; so the little ones are finished, too". And it promptly terminates the J and K loops, removing the tables from its memory.

Exactly what we want - but there are a couple of hidden gotchas that the user must know about when he gets into tricky coding routines.

The easiest one to spot is the situation where every loop has a different variable name. The first loop is, say, FOR A ... the next one, FOR B ... and the programmer continues through the alphabet with each loop. His idea is good: he can analyze how each loop has behaved, for each variable remains untouched for his examination. But each time he jumps out of a loop, the loop tables remain in memory, using up valuable stack or table space. He'd be much better off to give at least his outer loops the same variable name, and reclaim that space.

The second problem spot is a little more subtle, and an example would best illustrate it.

Here's a simple program to input a string, extract the individual words (eliminating single or multiple spaces), and print them:

```
100 INPUT S$          get the string
110 K=1               mark start-of-string
120 FOR J=K TO LEN(S$)
130 IF MID$(S$,J,1)<>" " GOTO 150     skip spaces
140 NEXT J
150 IF J>LEN(S$) GOTO 900
160 FOR K=J TO LEN(S$)
170 IF MID$(S$,K,1)=" " GOTO 200      scan to space or end
180 NEXT K
200 PRINT MID$(S$,J,K-J)
800 IF K<=LEN(S$) GOTO 120
900 END
```

The program works quite well, and isn't hard to follow. It should be noted that if either the J or K loops run to completion, the variable will have a value of LEN(S$)+1; this is intended and allowed for in lines 150 and 800.

Before we extend this program into catastrophe, let's note one thing: by the time the program reaches line 200, both the J and K loops will still be open most of the time - we "jumped out" of both of them. No real problem; when we go back to 120, the new FOR J= .. will cancel them both.

Now let's get into trouble. We may be writing a little ELIZA here, and want to check the word we've found against a table of keywords so as to pick a suitable reply. We'll assume a table of twenty keywords, and start to build a search loop. Replacing line 200, we start a new loop:

```
200 X$ = MID$(S$,J,K-J)      get word
210 FOR I=1 TO 20
  ......
```

Our loop is now three deep - J and K are still considered active, remember? No problem with three-level loops; we're still OK.

But here's where we might get clever and wreck everything. We need to preserve K - that's where our search for the next word will start. But J has served its purpose, and could be used again, right? Well .. let's see.

This table of 20 words is really a double table. It contains pairs of words such as "I","YOU", or "MY","YOUR". To make our computer talk we must spot a word from either column, and switch in the word from the opposite column (so that "I HAVE FLEAS" will become "YOU HAVE FLEAS"). So we need one more loop to search over the two columns.

Let's be clever and use J, since we have decided that it isn't needed any more at this point. We code:

```
220 FOR J=1 TO 2
230 IF X$=T$(I,J) THEN X$=T$(I,3-J): GOTO 400    swap word
240 NEXT J
250 NEXT I
400 PRINT X$;" ";                    repeat word
```

Suddenly everything stops working, and the world tumbles down around our program. What happened?

Let's stop and analyze. Just before executing line 220, the computer had three active loops, with variables J, K, and I. Now it reaches line 220, and what does it see? A loop based on J, the "biggest" loop! So what does it do? It cancels the K and I loops, of course, and starts a new J loop.

When we reach line 250, the computer sees NEXT I - but it no longer has an active FOR I= loop, and you get a NEXT WITHOUT FOR error notice.

The rule here is slightly more complex, but not too tough. If you use J as an "outer" loop variable, never use it for an inner loop. If we reversed I and J in the coding from 210 to 250, we'd have no problem. Try to think in terms of the hierarchy of loops, and you can make sure that a given variable is used only at its proper hierarchy level.

Let's try to put the rules together and create a tiny Eliza, polishing up some of the coding as we go. You'll have fun adding your own features to it.

```
100 DIM T$(1,4)        two by five array
110 DATA ME,YOU,I,YOU,MY,YOUR,AM,ARE,MYSELF,YOURSELF
120 FOR J=Ø TO 4
130 FOR K=Ø TO 1
140 READ T$(J,K)
150 NEXT K
160 NEXT J
170 INPUT S$
180 K1=1
190 FOR J=K1 TO LEN(S$)
200 IF MID$(S$,J,1)=" " THEN NEXT J
210 J1=J
220 IF J>LEN(S$) GOTO 900
230 FOR J=J1 TO LEN(S$)
240 IF MID$(S$,J,1)<>" " THEN NEXT J
250 K1 = J
260 X$=MID$(S$,J1,K1-J1)
270 FOR J=Ø TO 4
280 FOR K=Ø TO 1
290 IF T$(K,J)=X$ THEN X$=T$(1-K,J):GOTO 320
300 NEXT K
310 NEXT J
320 PRINT " ";X$;
330 IF K1<=LEN(S$) GOTO 190
~~340 GOTO 170~~ 340 PRINT "?"
900 GOTO 170
```

Note that the outermost loop is now always called J, the next down always K. I've tightened up the arrays to use the zero rows and columns to save memory; and the search loops are a little faster.

Even though the program is riddled with premature loop exits, there are no problems. Just observe a few simple rules, and you'll have efficient and trouble-free loops.

Computer City has announced some new products for PET.

Internal RAM expansion boards

16K	$395.00
32K	$495.00

Reset buttons $29.95

Machine language editor, assembler and disassembler package with 33 pages of documentation for $39.95

For more information contact Stu Bright at:

Computer City
1353 Portage Ave.
Winnipeg, Manitoba
R3H 0N1
(204) 786 3381

The PET® Gazette With A New Name

COMPUTE.

The Journal for Progressive Computing™

From: the new publisher of the PET® Gazette AUGUST 1, 1979

To: readers of the PET Gazette

During the past year, the PET Gazette has grown from a one page newsletter with a circulation of 50 to a multi-page magazine with a circulation of over 4,000. Due to this rapid growth the magazine has reached a crucial turning point. Len has been almost single handedly publishing the magazine each issue, and writing for it and many other magazines, as well as typing it, handling advertising placement, and followups, and new subscribers, and printers, and ... well you get the picture. The demand for the PET Gazette has begun to exceed his individual resources. Regardless of best intentions, the press of business problems left him falling behind in adding new subscribers, pressing his printer for delivery, and so on. Worst of all, he found himself losing the time needed for writing and programming.

The solution? Small System Services, Inc. has now acquired the PET Gazette. We're responsible for handling the many problems associated with bringing an active, energetic magazine from conception to your door. We're lining up an industry sensitive group of contributing editors; we're expanding the emphasis of the magazine. You'll still see the PET Gazette with a primary emphasis on the products (and problems) of Commodore Business Machines, Inc. But you'll see other 6502 products as well. And continuing features on such areas as Education, Business, New Products, and Reviews. You'll see an on-going illustrative column on maintenance and trouble shooting for a wide range of problems. And you'll get advice on program design and style. Plus new features to bring you more information from several viewpoints. You will most assuredly benefit from all these additions.

Your role as a reader? We'll continue to solicit your advice and written contributions. From our perspective, we'll shape up. The PET Gazette's next issue will be a massive super issue, Fall 1979, including a Christmas Buyers Guide, and general hints of what's to come. We're upgrading the quality of the magazine, and redesigning it for reading ease and practical use. This super issue of the New PET Gazette will be distributed free to all current subscribers. It will be followed by the first of our normal bi-monthly issues. As of the January/February 1980 issue the PET Gazette will carry an annual subscription price of $9.00 for 6 issues. The single copy newstand price will be $2.00. A third option, which we encourage where feasible, is a "personal/retail" subscription. You subscribe through us at the reduced annual rate of $7.50, and each of your prepaid issues will be included in the advance shipment to your local dealer with his standing order. You simply stop in and pick up your prepaid copy. We save the mailing costs, and you get your issue as fast as UPS reaches your dealer, and save a few dollars to boot.

But what happens to Len Lindsay after all this? Well, we've acquired the PET Gazette, not Len Lindsay. He'll give up the title of jack-of-all-trades while retaining the title of Senior Contributing Editor. He'll remain his outspoken self with no allegiance to anyone. We'll handle the business end, help coordinate the material from the newly added associate editors, etc. but best of all, Len will return to the more exciting aspects of this "business" ... writing, programming, and reviewing. The new expanded PET Gazette will continue to include all the help and information as previously, but will now also include articles and columns by the leading names in the field. Included will be specific help with PET Assembly Language Programming and good hardware help.

Stick with us. It's an exciting time for the new PET Gazette. Our title is "COMPUTE, the Journal for Progressive Computing.™" We plan to live up to the goals implicit in that statement. If you've sent in a donation check to Len since May 1, let us know if you wish it applied toward your new subscription (include a photocopy of your receipt from Len). If you are interested in our "Personal/Retail" subscription plan, include the name, address, and phone number of the dealer you'd like to subscribe through along with your check. Checks should be made payable to: "COMPUTE." Regardless of your decision now, you can expect to see the next issue of the new expanded PET Gazette in early October. Or stop by the PET Gazette booth (119-A) at the Boston Computer show Sept. 28-30. We hope you are as excited as we are about these improvements and additions to the PET Gazette. Thank you for your continued support.

Cordially,

Robert C. Lock President, Small System Services, Inc.

Len Lindsay Senior Contributing Editor, COMPUTE. The Journal for Progressive Computing™

PET® is a trademark of Commodore Business Machines, Inc.
"COMPUTE. The Journal for Progressive Computing" is a trademark of Small System Services, Inc.

900-902 Spring Garden Street
Greensboro, N.C. 27403
919-272-4867.